C/C++ Contributing Editor

The New C: Why Variable Length Arrays?

Randy Meyers

C meets Fortran, at long last.

In C99, the bounds of an array can now be a run-time expression. Such arrays are called variable length arrays or VLAs for short. VLAs can simplify storage management in a program and allow the use of the normal array notation even when the problem to be solved requires arrays to have different sizes at different times. To appreciate VLAs, you need to understand the problems in C90 that they remedy. To understand the problems and the way that VLAs work, you need to understand the relationship between pointers and arrays in C. This month's column concentrates on the above topics. Next month, VLAs will be presented in their entirety.

Pointers and Arrays

If you ask programmers to name the greatest strength of C, many of them will say pointers. There is good reason for this, since pointers in C have the following four powerful features:

1. Pointers in C are strongly typed. The type of a pointer includes the type of the object pointed to. Thus, if you dereference a pointer to **float**, you get a **float**.
2. Pointers in C are an orthogonal part of the type system. You can have pointers to pointers, arrays of pointers, and even pointers to entire arrays.
3. C has pointer arithmetic. You can add or subtract an integer from a pointer to get a pointer to a different object. You can subtract two pointers to get the integer distance (or offset) between the objects pointed to.
4. Pointers and pointer arithmetic are portable and shield the programmer from hardware differences. Unless a programmer is doing exceptional things, pointers and pointer arithmetic can be used without knowing the size in bytes of objects or their alignment requirements. (The exceptional things requiring such knowledge in C usually involve casts between pointers and integers or between different pointer types.) In fact, pointers in C are usually as portable as arrays in other languages. (The reason for this will be obvious shortly.)

While other languages have one or more of these pointer features, C was the first language to have them all. For example, some earlier systems implementation languages had pointer arithmetic, but that pointer arithmetic involved adding or subtracting the correct offset in bytes between objects. If C had adopted this model, then incrementing a pointer to an **int** would look like **p + sizeof(int)**. Instead, pointer arithmetic in C automatically multiplies the integer added or subtracted from a pointer by the size of the object being pointed to. When two pointers are subtracted, the result is automatically divided by the size of the objects. Without the automatic multiplying and dividing in C pointer arithmetic, programmers would have a little more work to do, have a few more chances for error, and have a needless temptation to write non-portable code. (Lazy programmers might tire of typing **sizeof (int)** when they could just type **4**.)

One of the consequences of C's pointer arithmetic was that arrays and pointers became interrelated concepts. In the definition of C, the **[]** operator is defined in terms of pointer arithmetic: **a[i]** means the same as ***(a + i)**. In turn, pointer arithmetic is defined in terms of the storage layout of arrays. If a pointer **p** points to an element of an array, then adding **n** to the pointer makes it point at the **n**th element following the original one. The equivalence of indexing and pointer arithmetic answers one of the common questions asked by new C programmers, "Why do array indexes start at zero rather than one?" If pointer **p** points at the first element of an array, then **p + 0** still points at that first element. Since an array name is just a pointer to the first element of the array when used with the index operator, and since indexing is by definition pointer arithmetic, **a[0]** must reference the first element of the array **a**.

Pointers are also very important to the processing of arrays. Since arrays cannot be passed by value to a function, a function that operates on an array is passed a pointer to the first element of the array. The language facilitates this by automatically rewriting the type of a parameter declared to be an array into a pointer to the array's element type. An array name used as an argument to a function is automatically converted to a pointer to the first element of the array. A pointer may be indexed by the **[]** operator since pointer arithmetic is the same for both arrays and pointers. The relationship between arrays and pointers is such that it really doesn't matter whether code operating on an "array" is really dealing with an array or a pointer to the first element of an array.

Problems with Arrays

Given that pointers are one of the strongest features of C, and that arrays are closely tied to pointers, it is surprising that C90 arrays are somewhat weak compared to other languages. The source of this weakness is two places in C90 where the size of objects must be known at compile time. Interestingly, these places hardly affect single dimension arrays, but greatly impact multidimensional arrays.

The first place is that the size of all types must be known at compile time. (There are incomplete types, not discussed further in this article, whose size need not be known at compile time, but their use is so restricted that you can neither fetch or store a value of incomplete type.) This means that you cannot declare an array whose size varies with the needs of the program as it runs. However, if you only need a single dimensional array, you can get the effect of an array with variable bounds by

dynamically allocating it. For example, the following dynamically allocates an “array” of **n** floats and initializes all of the elements with **1.0**.

```
float *a;
a = malloc(n * sizeof(float));
for (i = 0; i < n; ++i)
  a[i] = 1.0;
```

Note that even though **a** is a pointer, it can be indexed like an array using the **[]** operator because of the definition of **[]** in terms of pointer arithmetic. This approach works well for single dimensional arrays because, even though we have to dynamically allocate and free the storage ourselves, we can pretend that our pointer is an array in the rest of the code.

Unfortunately, this does not work as well for multidimensional arrays, which brings us to the to the second place where C90 requires the size of objects to be a compile-time constant: pointer arithmetic. The above code works because we do not need to know the size of the entire array to index into it, only the size of an element. However, if you have a two-dimensional array with bounds **m** and **n**, **int a[m][n]**, that means that **a** is an array of **[m]** elements each of which is an array of **[n] int**s. Evaluating the expression **a[i]** requires knowing the size of the element type, which is the size of an array of **[n] int**s. Since **n** is a run-time expression, the size of the array that is an element of the multidimensional array is a run-time expression, and it becomes impossible to calculate the first index operator given the restrictions in C90. The workaround in C90 for this problem is to take over not only the storage management of the multidimensional array, but its index calculations as well. The code looks like:

```
float *a;
a = malloc(m * n * sizeof(float));
for (i = 0; i < m; ++i)
  for (j = 0; j < n; ++j)
    a[i*n + j] = 1.0;
```

Note that uses of this “multidimensional array” do not look very natural. Instead of having two indexes for the two-dimensional array, we only have one. We manually have to multiply the “first” index by the number of elements in the second dimension, since each index value of the first dimension means skipping over the number of elements in the second dimension. The index expression looks ugly and gets uglier if there are more than two dimensions. In a three-dimensional array with bounds **[L][M][N]**, the first index is multiplied by **M*N** and the second index is multiplied by **N**. The C90 requirements for compile-time constant array bounds results in exactly the sorts of manual multiplies, verbosity, and opportunities for error that C so cleverly avoided in pointer arithmetic.

Enter Variable Length Arrays

C99 remedies these problems by adding variable length arrays to C. Given that pointers are important to the processing of arrays, C99 also adds pointers to variable length arrays and permits pointer arithmetic where the size of the object being pointed to is only known at run time. The **sizeof** operator is evaluated at run time if you apply it to a variable length array. (It is still evaluated at compile time if its argument is not a VLA.)

These changes mean that the natural array notation can be used even if the bounds of even a multidimensional array are all run-time expressions. Similar to previous examples,

```
void f(int m, int n)
{
  float a[m][n];
  int i, j;
  for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
      a[i][j] = 1.0;
}
```

The correct amount of storage for a VLA is automatically allocated when the block containing the array is entered and the declaration of the VLA is reached; the storage is automatically deallocated when leaving the block. Thus, VLAs can simplify storage management of programs since some uses that required manual use of **malloc** and **free** in C90 can be replaced by VLAs. For example, instead of code like:

```
void process(int n)
{
  // Set up a buffer of n characters
  char *b = malloc(n);
  // do the work
  ...
  // free the buffer
  free(b);
}
```

you could write:

```
void process(int n)
{
  // Set up a buffer of n characters
  char b[n];
  // do the work
  ...
}
```

The addition of variable length arrays to C99 finally gives arrays the same flexibility and elegance that pointer types in C have traditionally had. The compiler now automatically handles the storage management and the complexities of index calculations of arrays whose bounds are not compile-time constants. The result is that programmers can use the natural array

notation even when the size of arrays is only known at run time. Next month's column will cover the full semantics of variable length arrays and pointers to variably length arrays, including variable length arrays that are function parameters.

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com***.